Capstone Courseware, LLC
1 Washburn Place
Brookline, MA 02446
877-227-2477
capstonecourseware.com

Billing4-6

Starter code: **AmicaJavaTraining/Projects/Billing/Billing4**

Submit as: **CompletedWork/Billing/Billing5 (Derby)**
or .../Billing6 (MongoDB)

In this workshop you'll complete a reorganization of the Billing application's persistence system to support not only file formats, but databases. Specifically you will add support either for the Derby RDBMS or for the "noSQL" database MongoDB.

Though a great deal of the work is the same for either database, it will not be possible to support both at once. This comes down to a drawback of Spring Data's auto-configuration, that simply by their presence on the class path the libraries that support JPA and MongoDB will clash in attempting to set up their infrastructure. So you'll choose one path or the other – and, if you have time, you can always pursue the alternative database in a second Maven project.

To complete the databased application, you'll do the following:

- Re-connect the **Billing** class to file-based data, by adapting the existing **Parser** system to a new, more general interface that supports **Persistence** of a wider range of types. You'll be able to bring the (now-dormant) **BillingIntegrationTest** back on line at this point.
- Annotate the **Customer** and **Invoice** classes to support binding to your database of choice, and develop Spring Data repository interfaces to support persistence operations over these two types.
- Build a persistence service that uses the repositories to load data, save data, and act as a write-through cache. (The **Billing** class was acting as a cache before, but we want its job to be more narrowly defined as a sort of query engine, and to separate out the caching capability along with persistence generally.)
- Build a **Migration** utility that can populate your database with records from the existing CSV files, and an application that carries out this migration.
- Rebuild the **ReporterIntegrationTest** to run against a system of Spring components that includes your new persistence component and the running database.

Starter Code

The starter code in **Billing4** moves the application in the direction of a Spring Data application, in a few ways:

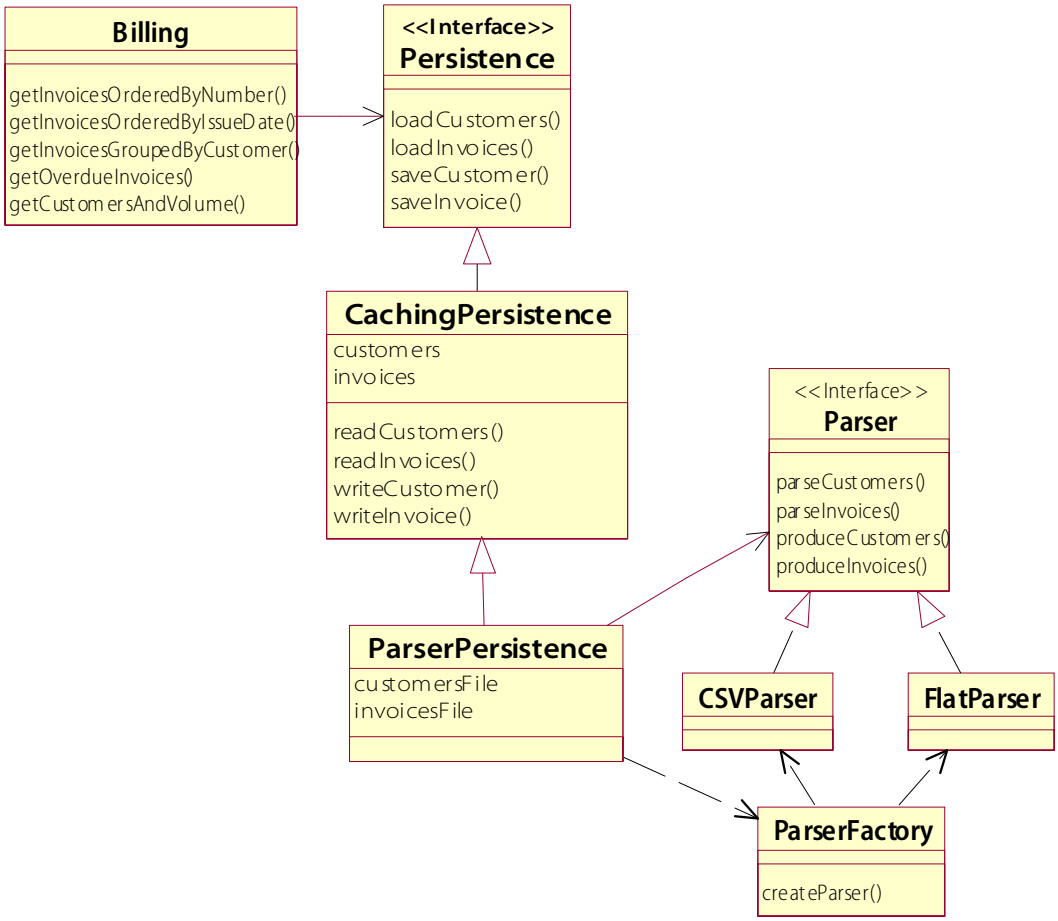
- The POM includes the **spring-boot-starter** and **spring-boot-test** dependencies.
- The **Billing** and **Reporter** classes are now Spring **@Component**s. The overloaded constructors that supported different configured and programmatic means of creating the objects have been replaced with a single constructor for each class that takes its collaborating objects as parameters.
- **Billing** now depends on a new, more generalized **Persistence** interface. Where a **Parser** is dedicated to persistent stores that can be read and written one line of text at a time, **Persistence** works at a higher level that can support databases, and even other file formats that don't happen to organize records on single lines of text, such as XML or JSON:

```
public Map<String,Customer> getCustomers();
public Map<Integer,Invoice> getInvoices();
public void saveCustomer(Customer customer);
public void saveInvoice(Invoice invoice);
```

- The **Invoice** class no longer stores its **paidDate** as an **Optional**, but just as a **LocalDate** that might be **null**. This is a concession to the fact that neither JPA nor MongoDB yet offer support for optionals. Constructors have been adjusted, and a getter and setter method added, so that callers still see the property as an optional, and as you develop the application to support a database, the data-binding library that's brought into play will see the raw **LocalDate** and will serialize that.
- The tests for **Billing** and **Reporter** that use the configuration manager are removed. The **ParserFactory** still supports the configuration manager, and the **ParserFactoryConfiguredTest** is still there.
- The **BillingIntegrationTest** is **@Disabled**, since we don't yet have a complete system of Spring beans to integrate into a testable system. **ReporterIntegrationTest** is replaced by a base class **ReporterIntegrationTestBase** that will be subclassed to create integration tests for specific databases.

Design

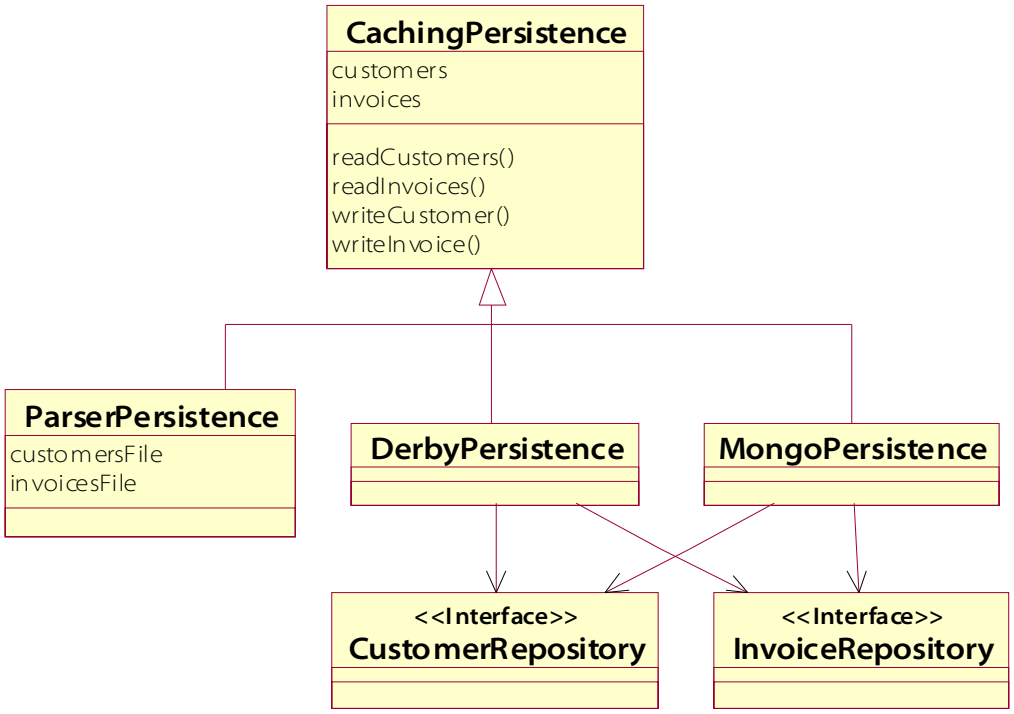
The new application mostly fits together as it has in earlier versions. The big difference is in the relationship between **Billing** and a chosen **Parser** – or with a **Persistence** object that doesn't do any text parsing at all.



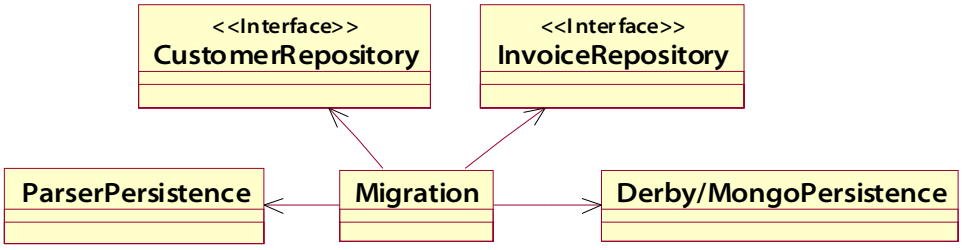
The **ParserPersistence** class will function as an Adapter between the existing parser-based implementations and the requirement for higher-level interactions (such as "load all customers" vs. "parse one customer from a string").

In doing so, it will take advantage of a base class **CachingPersistence** that captures loaded data as maps in memory, makes it available to callers, and updates those maps any time an object is saved. This is an application of the Template Method design pattern, as the base class can implement this write-through caching strategy in one place, while allowing derived classes to implement what it will call the "reading" and "writing" responsibilities as helper methods specific to a given persistent store.

Then other **CachingPersistence** subclasses are possible that will connect to databases. You'll implement either **MongoPersistence** or **DerbyPersistence**. The difference is just a few lines of code, as the Derby component must make its **load** method **@Transactional**.



The **Migration** component can then take advantage of one persistence object as its source and another as its target, and carry out a migration or replication scenario simply by loading the source and saving all records to the target:



Maven Dependencies

1. Start by adding the appropriate Maven dependencies to your POM, to support Spring Data for the database you'll use. XML fragments are provided for each of these options, in files **DerbyDependencies.txt** and **MongoDependencies.txt**. In each case, you can replace the **string-boot-starter** dependency with the contents of the text file, because it is a transitive dependency of either of the **spring-boot-starter-data-???** Artifacts.

   For Derby this involves a few dependencies, mostly because you use Spring Data JPA, which in turn can support any relational (SQL) database, so you have to specify the Derby JDBC driver:

```
<dependency>
  <groupId>javax.xml.bind</groupId>
  <artifactId>jaxb-api</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
  <groupId>org.apache.derby</groupId>
  <artifactId>derbyclient</artifactId>
</dependency>
```

   For Mongo it's simpler, because Spring Data Mongo is more narrowly targeted:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-mongodb</artifactId>
  <version>2.5.4</version>
</dependency>
<dependency>
```

CachingPersistence and ParserPersistence

2. Create an abstract class **com.amica.billing.db.CachingPersistence** that implements the **Persistence** interface. (Do be sure to stick with the package specifications of each new type as you work through these instructions. In ordinary Java this is almost always just a style choice, but in Spring component scanning it matters!)
3. Add the Lombok **@Getter** annotation to the class.
4. Define a map with strings as keys and **Customer** objects as values, as a **protected** field called **customers**.
5. Define a **protected** field **invoices** that is a map with integer keys and **Invoice** objects as values.

   Notice that, thanks to the **@Getter** annotation, the **getCustomers** and **getInvoices** methods on the interface are now implemented! You'll write the other two by hand.
6. Declare four abstract methods – these are the helper methods as described in the Template Method pattern. Derived classes will implement them, and in a moment you'll write the template methods that rely on them:

```
protected abstract Stream<Customer> readCustomers();
protected abstract Stream<Invoice> readInvoices();
protected abstract void writeCustomer(Customer customer);
protected abstract void writeInvoice(Invoice invoice);
```

7. Create a public method **load** that has **void** return type and takes no parameters. Implement it to call **readCustomers** and to compile the stream of customer objects into a map, and capture that map as your **customers** field – very much as you did before in the **Billing** class, and you may want to refer to code in **Billing3** for this. Now, you are not opening a file, but taking a stream from a helper method; but you still need to use try-with-resources because that stream, whether file-based or from a Spring Data repository, must be closed when you're done.
8. Do the same with invoices, so that your **invoices** map is initialized as well.
9. Create a **saveCustomer** method, taking a **Customer** parameter and returning nothing. Implement it to **put** the customer (new or updated, we don't care) into the map, and then call **writeCustomer** to let the subclass store it.
10. Do the same for invoices in a new method **saveInvoice**.
11. Create a new class **com.amica.billing.parse.ParserPersistence**,and make it extend **CachingPersistence**.
12. Annotate the class as a **@Component**.

13. Give it string fields **customersFile** and **invoicesFile**.
14. Annotate each as an injectable **@Value** based on a Spring property of a name such as "ParserPersistence.customersFile". Remember to enclose the property name, first in quotes and then in `${}`, so that Spring sees it as an expression to be resolved, rather than a literal value to be used directly.
15. Also place the Lombok **@Setter** annotation on each of these fields. This will give tests the option of setting the values programmatically.
16. Give the class a field **parser**, of type **Parser**.
17. Override the **load** method, and annotate your override as a **@PostConstruct** hook. Implement this method to initialize **parser** based on a call to **ParserFactory.createParser**, passing either of the injected file names. Again, the **Billing.java** file from **Billing3** may be helpful for these next few steps.
18. Implement the four helper methods defined by the base class. For each, you will either read or write the complete contents of one of the two files. For example, **readCustomers** will open the **customersFile**, use the configured **parser** to read its contents as a stream of customers, and return that stream to the base-class template method. Do not use try-with-resources here, because we want to return the stream to the template method, and let it process and then close the stream when done. Do try/catch, however, and on an **IOException** log a warning and return an empty stream.

    Note that **readInvoices** will need to pass **customers** along to the **parseInvoices** method, and the base class reads customers first, so this will work.

    The **writeXXX** methods will re-write the entire contents of their target files, ignoring the method parameter – the specific object to save – and getting all of the values from the **customers** or **invoices** map instead. Do use try-with-resources here.
19. Open **BillingIntegrationTest** and remove the **@Disabled** annotation.
20. Remove the placeholder "return null" implementation of the **createBilling** helper method. Instead, create a new **ParserPersistence** object, and set its filenames as we used to pass them to the **Billing** constructor; that code has been left in place, now in comments, so you can grab the expression for each filename and pass it to the appropriate setter method. Then call **load** on the object.
21. Then create a new **Billing** object, passing your persistence object, and return that.
22. You should be able to run the test now, and it will load the same files as before, and it should be passed.

Entity Classes and Repositories

These instructions will vary based on your choice of database.

23. We're going to move away from the first/last name as a key for customers, and instead let the database generate an ID for each new record. Open the **Customer** class and ...

    - For Derby, annotate the class as a JPA **@Entity**. Also add an integer field **ID**, bearing annotations identifying it as the primary key for the resulting database table, and declaring that it should be a generated value. All symbols below should be imported from **javax.persistence**.

```
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
private int ID;
```

    - For Mongo, add a string field **_id**. MongoDB recognizes this field name and will generate an **ObjectId** value for each new customer "document."

24. Remove the **@AllArgsConstructor** annotation, and add an explicit constructor that takes the first and last names and the payment terms and initializes those fields.

25. Open the **Invoice** class. For Derby only, add the **@Entity** annotation. Here we're going to continue to use the invoice number as a "natural" key, so simply annotate this field as an **@Id** – but import the symbol from **javax.persistence** for Derby, and from **org.springframework.data.annotation** for Mongo.

26. The **customer** field will require some attention, too:

    - For Derby, we want this to be a foreign-key relationship to a separate table, so add the JPA annotation **@ManyToOne**.

    - For MongoDB, we want it to be a reference from one document to another, so add the **@DBRef** annotation.

27. Define an interface **com.amica.billing.db.CustomerRepository**, like this:

```
public interface CustomerRepository
    extends PagingAndSortingRepository<Customer,String> {
  public Stream<Customer> streamAllBy();
  public Customer findByFirstNameAndLastName(String firstName, String lastName);
}
```

28. Define an **InvoiceRepository**, similar to the customer one but using **Integer** instead of **String** for the key type, and without the method to find by first and last name.

29. Create the application class: either **com.amica.billing.db.derby.MigrateCSVToDerby** or **com.amica.billing.db.mongo.MigrateCSVToMongo**.This will ultimately carry data into the application from CSV files, but it also offers the earliest opportunity you'll have to test out your repositories and entity/document classes. Annotate the class appropriately as shown in the following listings:

```
@ComponentScan(basePackageClasses={CustomerRepository.class})
@EnableAutoConfiguration
@EnableJpaRepositories(basePackageClasses=CustomerRepository.class)
@EntityScan(basePackageClasses=Customer.class)
@EnableTransactionManagement
@PropertySource(value=
       {"classpath:DB.properties","classpath:migration.properties"})
public class MigrateCSVToDerby {
}
```

```
@ComponentScan(basePackageClasses={CustomerRepository.class})
@EnableAutoConfiguration
@EnableMongoRepositories(basePackageClasses=CustomerRepository.class)
@PropertySource(value=
       {"classpath:DB.properties","classpath:migration.properties"})
public class MigrateCSVToMongo {
}
```

What does all of this mean? This is (deep breath): a Spring configuration that enables Spring Boot auto-configuration (which will pick up the Spring Data JAR and set that up) and either JPA or Mongo Spring Data repositories from the **com.amica.billing.db** package, and loads injectable property values from the file **migration.properties.**

The Derby version of the class also explicitly scans for JPA entities and enables a transaction manager, which we'll need for some operations in a minute.

30. Define a **main** method that uses a try-with-resources block to call **SpringApplication.run**, passing the class as itself the configuration class.

31. In the **try** block, get the **CustomerRepository** bean from the context and assign it to a local variable, so you can make a few calls on it and test it out.

32. Print the results of calling **count** on the repository.

33. Create the supporting properties file **migration.properties**, in **src/main/resources**. Give it properties as follows – filenames that we'll use later to identify our source data …

```
ParserPersistence.customersFile=data/customers.csv
ParserPersistence.invoicesFile=data/invoices.csv
```

34. Create another file **DB.properties**, in the same folder. , for Derby define these properties (and you can get a starter for this from the **BowlingDB** project) …

```
spring.datasource.url=jdbc:derby://localhost/AmicaTraining;create=true
spring.datasource.driver-class-name=org.apache.derby.jdbc.ClientDriver
spring.datasource.username=Billing
spring.datasource.password=Billing
spring.jpa.database-platform=org.hibernate.dialect.DerbyDialect
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=none
```

… or for Mongo, just set the database name …

```
spring.data.mongodb.database=billing
```

35. Start your database of choice, using either the first or second set of commands below:

```
cd your-location-for\Tools\Derby10.8\bin
startNetworkServer
```

```
cd your-location-for\Tools\Mongo5.0
md data
bin\mongod --dbpath data
```

36. Now run the application. It should initialize Spring Data, connect to the running database server, create the necessary schema and tables, or database and collections, and run a simple query for customers, returning a count of zero.

    This is a big milestone, because it proves that all of that infrastructure is in place and correctly configured. A lot can go wrong here, and if it doesn't run correctly, take a look through the console output and look for likely causes – connection failures, beans not defined, etc. See the following page for some troubleshooting advice.

Troubleshooting

The first time Spring Data JPA runs and parses the entity classes, it will generate a relational schema. If you miss something in how you annotate those classes, and then make a change, it may not work with the now-existing schema. Mongo has a much looser approach, and by default there is no formal schema. This means fewer runtime exceptions, but sometimes you get inconsistencies in the data, especially if you change the document classes from one run to the next.

Troubleshooting for JPA and Derby

If for any reason you want a clean slate, you can set the spring.**jpa.hibernate.ddl-auto property** to "create-drop" and it will regenerate on each run. Don't leave it at this setting, though, as this means the data will be destroyed every time you run!

There is also a SQL script **data/removeDB.sql** that you can run with the **ij** utility.

```
cd your-location-for\Billing4
your-location-for\Tools\Derby10.8\ij data/removeDB.sql
```

Or, run **ij** as an interactive SQL terminal, and you can check on the results of running the application – for example:

```
cd your-location-for\Tools\Derby10.8\bin
ij> connect 'jdbc:derby://localhost/AmicaTraining';
ij> set schema Billing;
ij> describe customer;
ij> select * from customer;
ij> select * from customer where ID=1;
```

Troubleshooting for MongoDB

The easiest way to clean out the database on Mongo is to stop the server (Ctrl-C is fine), delete and re-create the **data** folder, and run the server again.

You can inspect the MongoDB database using the **mongo** client – for example:

```
cd your-location-for\Tools\MongoDB5.0\bin
mongo
> show databases
> use billing
> show collections
> db.customer.find()
> db.customer.count()
```

37. Once the application is working, you might want to explore use of the repository interface a little. Try creating a new **Customer** object – just using Java **new** and whatever names and payment terms – and passing it to the **save** method on the repository. Then call **count** again and see that it's one. Run the application again and see that more customers are created, and this is okay since they have generated, unique IDs. Try calling **findAll** and printing out the results. Try calling **findByFirstNameAndLastName**: if you get an exception here it might be that the name doesn't match, or that you asked for a customer you created multiple times and Spring Data can't find just one result. Note that you can clear the table or collection by calling **deleteAll**.
38. Get the **InvoiceRepository** bean, and exercise that, in a similar way. You can use the customer repository to get a specific customer, and then create a new invoice for that customer and save it with the invoice repository. Try **findById** and pass the invoice number you assigned … etc.

Persistence Service, Application, and Migration Component

39. Create a persistence-service class, either **DerbyPersistence** or **MongoPersistence**, in the same package as your application class. Make it extend **CachingPersistence**.
40. Annotate as a **@Component**.
41. Give your class fields of type **CustomerRepository** and **InvoiceRepository**, and a constructor that initializes them both.
42. Override the **load** method, and annotate your override with **@PostConstruct**. Just call the superclass method from your override; this will cause the class to "auto-load" when configured as a Spring component.
43. Implement the four helper methods as required by the base class. These will all be simple pass-through methods: **readCustomer** will call **streamAllBy** on the customer repository, and return the results; **writeCustomer** will pass the given customer to **save** on the repository; and similarly on the invoice side.

44. Now, for Derby only, you need to do a bit more. Spring Data JPA will insist that you call a repository method from within a database transaction – owing to the deferred execution that the stream implies and Spring not wanting to leave a database connection hanging. To do this in most cases just means annotating the appropriate method as **@Transactional**, but it's a little trickier when we're trying to auto-load from a post-construct hook, because the transaction system may not yet be up and running when a given bean is configured.

    So, first, define an autowired field **txManager**, of type **PlatformTransactionManager**. This will assure that the **load** method is not called until that system is ready to create a transaction for us.

    In your **load** override, instead of calling **super.load** directly, do this:

```
@Override
@PostConstruct
public void load() {
    TransactionTemplate template = new TransactionTemplate(txManager);
    template.executeWithoutResult(status -> super.load());
}
```

45. In your application class, comment out the code in your **try** block, and instead write some code that gets the bean of your persistence-service type and exercises it. You should be able to call **getCustomers** right off the bat, and if you print out the map you'll see the customers you created earlier. Do the same with **getInvoices**. Try saving a customer or invoice this way, too.
46. Create a **@Component** class **com.amica.billing.db.Migration** and give it four, autowired fields: a reference to each of the two repositories, a **ParserPersistence** called **source**, and either a **DerbyPersistence** or a **MongoPersistence** reference called **target**.
47. Define a method **migrate** that takes no parameters and returns nothing. Implement it to call **source.getCustomers** and, for each element, call **target.saveCustomer**.
48. Remove or comment out all of your code in the **main** method's **try** block. Instead, get the **Migration** bean from the context, and call **migrate** on it.
49. Then get the customer repository bean, and print out its **count**. In fact, print it out both before and after the call to **migrate**.

50. Run the application. You'll hit an issue here: an unsatisfied dependency on **ParserPersistence**. Can you see why this isn't working yet?

    Here is a hint: it has to do with the scope of component scanning, which you initially defined to be based in the **com.amica.billing.db** package ...

```
@ComponentScan(basePackageClasses={CustomerRepository.class})
```

    This was enough to bring in your persistence service, and now your **Migration** component. But it's not a wide enough net to include **ParserPersistence**, which lives in **com.amica.billing.parse**.

    You can fix this by adding **ParserPersistence.class** (or in fact any type from that package) to the list of basePackageClasses**.**

51. Try it again. You should see that the customer count increases by 13 as a result of the call to **migrate**. This is the bean asking the parser-based persistence "source" to load the 13 customers defined in **customers.csv**, and saving them all to the repository-based persistence "target."

52. We'll want more reproducible results going forward, so in the **migrate** method, before transferring customers from source to target, call **deleteAll** on the invoice repository, and then call it on customer repository. (You could get away with just the customer repository, for now, on Mongo; but you'll need both of these eventually. On Derby you need them now, because otherwise you'll get errors trying to remove customers with invoices that refer to them still in the database.)

53. Run the application again and you should see the count go from whatever it was to exactly 13, and on later runs it will start at 13 and end at 13.

54. Add code to the **migrate** method to copy all invoices from source to target. Test this out in your application as well.

55. After deleting all customers and invoices, and before transferring records, call **load** on both source and target services. This won't matter as you're running your application right now, but it will make the **Migration** bean more reliable later, in contexts where it and the source and target might all be used repeatedly and after changes to the underlying data and/or the in-memory caches.

Reporter Integration Test

56. Finally, you can re-assemble the integration scenario to test the whole system, now resting over a database rather than files. Review the **ReporterIntegrationTestBase**, and see the utility methods and test cases from before, but notice the old configuration is gone. Instead we have autowired **Billing** and **Reporter** dependencies and a **@BeforeEach** method that focuses on setting up the output folders for the reporter under test.
57. Create a subclass of this class: either **ReporterDerbyIntegrationTest** or **ReporterMongoIntegrationTest**, but in the same package **com.amica.billing** as the base class.
58. Give your class a public, static class called **Config**.
59. Annotate the outer class to **@ExtendWith** the **SpringExtension,** and as a **@SpringBootTest** with **classes** set to your **Config.class**.
60. Annotate your nested **Config** class like this, for Derby:

```
@ComponentScan
@EnableAutoConfiguration
@EntityScan(basePackageClasses=Customer.class)
@EnableTransactionManagement
@PropertySource(value={"classpath:test.properties",
    "classpath:migration.properties"})
public static class Config {
}
```

… or like this, for MongoDB …

```
@ComponentScan
@EnableAutoConfiguration
@EnableMongoRepositories
@PropertySource(value={"classpath:test.properties",
    "classpath:migration.properties"})
public static class Config {
}
```

So, we're using component scanning – ready to pull in the **Billing** and **Reporter** components, along with our new persistence system, Spring Data repositories, entities, etc.

61. Add a file **test.properties** to **src/test/resources**. In it, define **Reporter.outputFolder** to the value "reports", and **Reporter.asOf** to "2022-01-08".

62. Also, copy the contents of your **DB.properties** into **test.properties**, and then change the username/password (for Derby) or database name (for Mongo) to "BillingTest". This will assure that the integration test doesn't interfere with the migration application, and vice-versa.

63. Try running your test now. You'll hit a problem – a different sort of unsatisfied-dependency issue in that **Billing** autowires a Persistence provider … and you have two of them! Spring doesn't know which one you want.

    There are a handful of ways to solve this: Spring profiles … demoting one of the classes so it's no longer a scannable component, and setting it programmatically …. Narrowing the scope of component scanning … and more. We're actually going to need the **ParserPersistence** component to be configured, later on, or narrowing our scope with **basePackageClasses** might be the best choice.

64. Instead, add the **@Primary** annotation to your persistence service. This will let us use both components later, while clarifying for Spring that, where **Billing** wants any one **Persistence** bean, your persistence service should be preferred.

65. Test again, and if all is configured correctly … well, some of the tests will pass! but not all of them. Essentially, the tests that check reports directly from the starter data set should succeed – and that's pretty cool, right? So we're drawing data from the database, the **Billing** query methods are distilling it in some way, and the **Reporter** is formatting out the reports.

    But, as soon as one of the tests that involves changing the data is run, that test may succeed, but most later tests will fail, because the database now holds unexpected data, which will lead to unexpected results. For example once **testPayInvoice** runs, tests that list invoices in any way will be wrong, because one of the invoices now has a paid date that we didn't think it would have.

66. How you solve this will depend on the database you're using: with Derby, it will mean running a migration **@BeforeAll** tests, making those mutating test methods **@Transactional**, and re-loading the persistence service **@BeforeEach** test. For Mongo it will mean running a migration and re-load **@BeforeEach** – simpler to code, but slower to run.

The remaining instructions are all database-specific. For Derby:

67. Define a **@BeforeAll** method on the outer class. In it, use try-with-resources to create an **AnnotationConfigApplicationContext** based on your own **Config** class.
68. In the **try** block, get the **Migration** bean, and call **migrate**.
69. Give the outer class an autowired field of type **DerbyPersistence**.
70. Override **setup**, and annotate your override as a **@BeforeEach** method, and you'll have to state that it **throws IOException**, because the superclass method does. In this method, call the superclass method and then call **load** on your **DerbyPersistence** field.
71. Override **testCreateCustomer**, and annotate your override both as a **@Test** and as **@org.springframework.transaction.annotation.Transactional**. In your method, just call the superclass method.
72. Do the same for **testCreateInvoice** and **testPayInvoice**.
73. Run the test again, and it should pass – all methods, now, because you reset the data to start out, and whenever a method makes a change, that change is rolled back by the test context after the test executes.

For MongoDB …

74. Define an autowired field of type **Migration** on your test class.
75. Override **setup**, and annotate your override as a **@BeforeEach** method, and you'll have to state that it **throws IOException**, because the superclass method does. In this method, call the superclass method and then call **migrate** on this object.
76. Run the test again, and it should pass – all methods, now, because you reset the data completely before each test.

If you have additional time, a great next step would be to create a subclass of **Billing** that takes better advantage of Spring Data's derived query methods. Inject references to the repositories, and add methods to them that let the database server perform the query logic – such as **InvoiceRepository.streamByOrderByNumberAsc**, **streamByOrderByDateAsc**; **CustomerRepository.streamAllByTerms**, etc.